Model checking adaptive software with featured transition systems

Cordy, Maxime; Classen, Andreas; Heymans, Patrick; Legay, Axel; Schobbens, Pierre

Published in:
Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)

DOI:
10.1007/978-3-642-36249-1

Publication date:
2013

Document Version
Peer reviewed version

Model Checking Adaptive Software with Featured Transition Systems

Maxime Cordy*[1], Andreas Classen[1], Patrick Heymans[2], Axel Legay[3], and Pierre-Yves Schobbens[1]

[1] PreCISE Research Center, University of Namur, Belgium.
{mcr,acs,pys}@info.fundp.ac.be

[2] PreCISE Research Center, University of Namur, Belgium.
INRIA Lille-Nord Europe – Universit Lille 1, France.
LIFL – CNRS, France.
phe@info.fundp.ac.be

[3] INRIA Rennes, France.
Aalborg University, Denmark.
University of Liège, Belgium.
axel.legay@inria.fr

Abstract. We propose to see adaptive systems as systems with highly dynamic features. We model as features both the reconfigurations of the system, but also the changes of the environment, such as failure modes. The resilience of the system can then be defined as the fact that the system can select an adequate reconfiguration for each possible change of the environment. We must take into account that reconfiguration is often a major undertaking for the system: it has a high cost and it might make functions of the system unavailable for some time. These constraints are domain-specific. In this paper, we therefore provide a modelling language to describe these aspects, and a property language to describe the requirements on the adaptive system. We design algorithms that determine how the system must reconfigure itself to satisfy its intended requirements.

1 Introduction

Our society increasingly entrusts computerized systems with complex and critical tasks. These systems have to be adapted, or adapt themselves, to a rapidly evolving environment, while accomplishing their tasks reliably. Due to the short reaction times required, some of these adaptations have to be performed automatically, leading to *self-adaptive* systems. Such systems are usually architected in two levels: The base level manages the basic tasks of the system. It has a simple design that allows rapid response times, but does not allow to respond

* FNRS Research Fellow

to exceptional conditions. For instance, a satellite control system is in charge of maintaining the attitude of the satellite so that the solar panels face the sun. It must react rapidly when the satellite starts to spin. On the other hand, the adaptive level can detect a change of conditions, and reprogram the base system to adapt to these new conditions. Again, the base system is efficient, but often not exhaustive. For instance, when entering the shadow of a planet, the system will be adapted to give priority to the orientation of the data transmission antenna.

In this paper, we propose to model such adaptation by the notion of *feature*, borrowed from product lines engineering. Classically, a feature is an added functionality to the system, that responds to a (new) need of the customer. Here, a feature can also be an adaption to environmental conditions. For uniformity, we also model such evolutions of the environment (in which we might include some parts of the system itself) as special "features". They can be failures (in which case the associated behaviour describes the failure mode and effects), increase of power of an attacker (in which case the associated behaviour describes the modus operandi of attacks), etc.

In some cases, preserving the functionality of the system is not possible, e.g. in the presence of severe failures. Therefore the requirements need to allow for degraded functionality in such cases, and thus our requirements logic (Section 4) also should include dependency on features. For instance, when a solar panel is damaged, the satellite is allowed to shut down some of its non-prioritary facilities (e.g., observation of aurora borealis) to preserve its vital functions (e.g., avoiding falling on Earth) instead. We thus define *resilience* as the capacity to ensure such conditional requirements in presence of a changing environment (at end of Section 4).

More concretely, our contributions are the following: we propose in Section 3 a fundamental framework, called A-FTS, to model the evolution of both the environment and the adaptive system, and in Section 4 AdaCTL, a temporal logic to describe the requirements on such a system. The next step is, naturally, to check whether the model satisfies the requirements in face of a changing environment, i.e., its *resilience*. This resilience-checking problem departs from the classical model-checking problem in several ways, and thus requires specific adaptations of the classical algorithms, presented in Section 5.

Finally, we compare our approach to extant work in Section 6. This paper only addresses modelling and checking the behaviour of an adaptive system. We briefly sketch the other tools needed for a more comprehensive approach to (self-)adaptive systems in Section 7.

2 Background

In order to make this paper self-contained, we first recapitulate essential definitions related to SPL modelling and verification.

Model checking is a well-known technique for verifying software-intensive systems against temporal properties. In a nutshell, given the model of a system M and a temporal property Φ, a model-checking algorithm determines whether or

not M satisfies Φ, written $M \models \Phi$. One may use *labelled transition system* (LTS) as such a model.

Definition 1 *An LTS is a tuple $(S, Act, trans, I, AP, L)$ where S is a set of states, Act is a set of actions, $trans \subseteq S \times S$ is a transition relation, $I \subset S$ is a set of initial states, AP is a set of atomic propositions, and $L : S \to 2^{AP}$ is a labelling function that associates every states with the set of atomic propositions satisfied by this state.*

We call an execution (or run) of the system an alternating sequence of states and actions. The semantics of an LTS, noted $[\![.]\!]_{LTS}$, is then its set of executions, that is,

$$[\![ts]\!]_{LTS} = \{s_0, \alpha_0, s_1, \alpha_1, \ldots, s_i, \alpha_i, \ldots \mid s_0 \in I \wedge (s_i, \alpha_i, s_{i+1}) \in trans\}. \quad (2.1)$$

In the context of SPLs, the model-checking problem becomes more complex as it requires to identify the exact set of products that do not satisfy a given property [17]. To answer it, one can model each product with an LTS and model-check each of them separately. However, for a SPL of n features, this would require $O(2^n)$ calls to a model checker. Given that distinct products of an SPL may have commonalities in their behaviour, there is a need for concise models and efficient algorithms able to distinguish between commonality and variability.

In this paper, we assume that the variability is captured in a *feature model*, features being atomic units of difference between products. A product is then uniquely defined by a set of features.

Definition 2 *Let F be a set of features. Then a product p is a subset of F, that is, $p \in \mathcal{P}(F)$ where $\mathcal{P}$ denotes the powerset.*

Several representations exist for feature models. Here, we remain at an abstract level and consider feature models independently of their representation. More precisely, we stick to the semantics of Schobbens *et al.* [47] and assume that a feature model defines a set of valid products, *i.e.*, a set of authorized combinations of features.

Definition 3 *A feature model is a couple*

$$d = (F, [\![d]\!] \subseteq \mathcal{P}(\mathcal{P}(F))) \quad (2.2)$$

where F is a set of features, and $[\![d]\!]$ is the set of valid combinations of features.

Given the similarities between different products, SPL modelling approaches aim to capture both their commonality and their variability in a compact manner [8, 22, 7, 17]. Most of them rely on the use of variability operators that determine which parts of the model may or may not be present in a given product; the non-variable parts being shared by all the products. For instance, we proposed *Featured Transition Systems* (FTS) as an extension of Labelled Transition Systems (LTS) meant to model the behaviour of SPLs [17]. FTS model design-time

variability of the system behaviour by labelling transitions (i.e., executions of actions) between two states of the system with Boolean constraints defined over the set of features. Then a given product is able to trigger a given transition if and only if its set of features satisfies the associated constraints. Such a constraint is called feature expression and is formally defined as follows.

Definition 4 *A feature expression exp defined over a set of features F is a total function*

$$exp : \mathcal{P}(F) \to \{\top, \bot\}. \tag{2.3}$$

For a given product p, $exp(p)$ returns $\top$ if and only if the features of p satisfies the constraints expressed by exp. In this case, we say that p satisfies exp. We denote by $[\![exp]\!] \subseteq \mathcal{P}(F)$ the set of products that satisfy exp and by $\top$ the feature expression such that $[\![\top]\!] = \mathcal{P}(F)$. In FTS, we use feature expressions to restrict the set of products able to execute a given transition [16].

Definition 5 *An FTS is a tuple $(S, Act, trans, I, AP, L, d, \gamma)$, where*

- *$S, Act, trans, I, AP, L$ are defined as in Definition 1,*
- *d is a feature diagram,*
- *$\gamma : trans \to \mathcal{P}(F) \to \{\top, \bot\}$ is a total function, labelling each transition with a* feature expression.

Thanks to the use of feature expressions, an FTS is a compact representation for a set of LTS, namely one per product. One can obtain the LTS modelling the behaviour of a given product by computing the *projection* of the FTS onto that product [17].

Definition 6 *The projection of an FTS fts to a product $p \in [\![d]\!]$, noted $fts_{|p}$, is the LTS $(S, Act, trans', I, AP, L)$ where $trans' = \{t \in trans \mid p \in [\![\gamma(t)]\!]\}$.*

The semantics of an FTS $\mathcal{M} = (S, Act, trans, I, AP, L, d, \gamma)$, noted $[\![fts]\!]_{FTS}$, is then defined as a function that associates a product with the semantics of its projection:

$$[\![\mathcal{M}]\!]_{FTS} : \mathcal{P}(F) \to (S \times Act)^{\omega} : \forall p \in [\![d]\!] \bullet [\![\mathcal{M}]\!]_{FTS}(p) = [\![\mathcal{M}_{|c}]\!]_{LTS} \tag{2.4}$$

Model checking an FTS against a property Φ comes down to distinguishing between the products that satisfy Φ and the ones that do not. This leads us to a new notion of satisfiability $\models_F$, which is not Boolean. Formally, if $\mathcal{M}$ is an FTS defined over a feature model d and Φ is a property, we have

$$(\mathcal{M} \models_F \Phi) = \{p \in [\![d]\!] \bullet \mathcal{M}_{|p} \models \Phi\} \tag{2.5}$$

Given the importance of features in SPLs, a suitable logic should include special operators that reason over them. For this purpose, we defined a featured extension of the *Computational Tree Logic* (CTL) [15]. It is called $fCTL$. Any formula defined in this logic has the form $[\chi]\Phi$ where χ is a feature expression and Φ is a CTL formula. Given an FTS $\mathcal{M}$ defined over a feature model d and a product $p \in [\![d]\!]$, p satisfies $[\chi]\Phi$ if and only if p does not satisfy χ or $\mathcal{M}_{|p}$ satisfies Φ.

3 Modelling the Behaviour of Dynamic SPLs

One way to model a dynamically adaptive software is to represent it as a set of static programs and transitions between them [51]. A transition between two programs models an adaptation of the system, which may be needed in case of changes in the properties of the environment. Since those programs are usually meant to satisfy the same set of high-level goals, they likely share commonalities in their structure and behaviour. Moreover, nowadays an increasing number of software systems are designed as product lines and we observe the emergence of *Dynamic Software Product Lines* (DSPLs) [33], especially in the mobile software industry. In order to cope with changing architectures and environments, these SPLs are equipped with the ability to change their set of features at runtime. In this context, we call *configuration* the set of features of a system at a particular point of time, and *reconfiguration* the process of altering the configuration of this system. The details of such reconfigurations are often hidden to the user who can only witness which *features* of the system have changed.

However, behavioural modelling approaches for SPLs like FTS do not consider that an SPL may have to adapt its behaviour due to unexpected changes in the environment. In other words, a given configuration is chosen and fixed through the whole execution of the system, which is thus completely dependent to the environment. Here, we propose a new modelling formalism that allows dynamic reconfiguration. More precisely, we introduce *Adaptive Featured Transition Systems* (A-FTS), an extension of FTS meant for modelling DSPLs. A-FTS explicitly represent the variability of both the system and its environment. The latter is defined as a set of features, such that an *environment feature* is a Boolean characteristic of the environment that may change over time and that the software has the ability to perceive. The capability of the software to execute a transition thus depends on both its features and those of the environment. For the system, we distinguish between *fixed*, non-mutable features and *adaptable* features. To capture the variability of both the system and the environment, we now define a feature model as a tuple

$$d = (F, F_s \subseteq F, F_a \subseteq F_s, [\![d]\!] \subseteq \mathcal{P}(\mathcal{P}(F))) \tag{3.1}$$

where F is the set of all the features, F_s is the set of the system's features, F_a is the set of the system's adaptable features, and $[\![d]\!]$ the set of valid configurations. We also assume that the system and the environment do not share any feature, so that we can define the set of the environment features (noted F_e) as $F \setminus F_s$. According to the above, we define a system configuration (resp. an environment configuration) as a subset of F_s (resp. F_e), and a (complete) configuration is the union of these two.

An adaptable feature may be enabled or disabled at some points during the execution. Unexpected variations in the environment may force the software to adapt its configuration so that it still works properly according to intended requirements. Given that objective, we consider that after the system executes a transition, it is able to observe the features of the environment that are enabled

at that time. According to the current configuration of the environment, the system may alter its own configuration so that it avoids failure or undesirable situations. Since we do not yet consider real-time constraints, we suppose that these reconfigurations are atomic and instantaneous. However, the system can be forbidden to reconfigure itself at some point because it must first terminate the execution of a given sequence of actions. In particular contexts, the environment can also be stable during a given period. The following formal definition of A-FTS takes all these considerations into account.

Definition 7 *An A-FTS is a tuple* $(S, Act, trans, i, AP, L, d, \gamma)$ *where*

- S, Act, $I \subseteq S$, AP, *and* $L : S \to \mathcal{P}(AP)$ *are defined as in Definition 1;*
- $d = (F, F_s, F_a, [\![d]\!])$ *is the feature model modelling the variability of both the system and the environment;*
- $\gamma : S \times Act \times S \to (\mathcal{P}(F) \times \mathcal{P}(F) \to \{\top, \bot\})$ *is a function that defines the transition relation.*

Note that unlike LTS and FTS, the transition relation is not defined as a set called $trans$. Instead, we use the function γ to encode symbolically which transitions exist, which products can execute them and how the configuration of the system and the environment evolve. More precisely, this function allows us to:

1. Determine whether of not the system can reach a state s' by executing an action α from a state s. If that is not the case then $\gamma(s, \alpha, s')$ is a function that returns a $\bot$ whatever the configuration of the system and the environment.
2. Restrict the set of configurations able to execute such a transition. Let us suppose that the the system can reach s' from s by executing α if and only if its features satisfy the feature expression exp. For any system's configurations $c \in [\![exp]\!], c' \in \mathcal{P}(F_s)$ and environment's configurations $e, e' \in \mathcal{P}(F_e)$, we have $\gamma(s, \alpha, s')(c \cup e, c' \cup e')$. This definition of transition relation is thus more general than in FTS.
3. Restrain how the configuration of the system and the environment can evolve after the execution of the transition. For instance, let us suppose that if the system moves from s to s' by executing α while in configuration c then it cannot change its configuration at all. To express that constraint we define that for any system's configuration c and environment's configurations e and e', we have for any c' that $\gamma(s, \alpha, s')(c \cup e, c' \cup e') = \bot$.

Note that γ must be defined such that only the adaptable features of the system may be enabled or disabled during runtime:

$$(c \setminus c') \cup (c' \setminus c) \not\subseteq F_a \implies \neg\gamma(s, \alpha, s')(c \cup e, c' \cup e') \tag{3.2}$$

for any $s, \alpha, s, c, c', e, e'$. Moreover, any reconfiguration of the system and the environment must ensure that the new configuration is valid (that is, it must satisfy the constraints of the feature model). Accordingly, the function γ must be such that:

$$c' \cup e' \notin [\![d]\!] \implies \neg\gamma(s, \alpha, s')(c \cup e, c' \cup e') \tag{3.3}$$

for any $s, \alpha, s, c, c', e, e'$.

Example 8 *An example of an A-FTS is graphically illustrated in Figure 1. It depicts a small behavioural model of an adaptive routing protocol inspired by the case study of Zhang et al. [51]. The system has one adaptable feature* encryption*, which leads to two possible configurations. Similarly, the environment has one feature* safe *that may or may not be enabled. Initially, the system is in state* `ready`*. Once it receives a message, it enters the state* `received`*. Then, it routes the message and enters either* `routed-safe` *or* `routed-unsafe` *depending on whether the environment is safe or not. This information is captured by the environment's feature* safe*. In our graphical representation, we model these restrictions by writing the feature expression that must be satisfied by the current configuration of the system and the environment for the transition to be executable. Formally, we make use of the transition relation γ for defining those restrictions. For instance, we know that the transition between* `received` *and* `routed-unsafe` *cannot be executed in a safe environment; accordingly, we define γ such that*

$$\neg\gamma(\texttt{received}, \texttt{route()}, \texttt{routed-unsafe})(c \cup e, c' \cup e') \tag{3.4}$$

where $safe \in e$ and for any e',c,c'. In order to increase readability, we do not explicitly represent the whole definition of the function γ.

If the environment is safe then the system simply sends the message, reaches the state `sent-safe` *and ends up going back to the state* `ready`*. If the environment is not safe and if the system's feature encryption is enabled, then the system encrypts the message and sends it afterwards. Otherwise, it sends the message unencrypted. In every case, the system eventually returns to state* `ready`*. In our graphical representation, the set of atomic propositions satisfied by a given state is written directly below it.*

Additionally, we define that once the system has routed the message it cannot change its configuration until it reaches state `ready` *again. Similarly, we suppose that the feature of the environment are stable between the routing and the sending. We capture those restrictions by means of the transition relation γ. For instance, we have*

$$\gamma(\texttt{routed-safe}, \texttt{send()}, \texttt{sent})(c \cup e, c' \cup e') \Leftrightarrow (c = c' \wedge e = e'). \tag{3.5}$$

for any c, c', e, e'. A property of interest for this system is that while the environment is unsafe, no package is sent before it is encrypted. Further in the paper, we introduce a new logic able to express such properties.

As mentioned in Section 2, an (infinite) execution of a transition system is defined as an alternating sequence of "states" and actions. Unlike LTS and FTS, the concept of state in A-FTS does not only refer to the state of the system itself, but also to its configuration as well as that of the environment. In order to avoid ambiguity, we call that a *macrostate*.

Definition 9 *Let $\mathcal{M}$ be an A-FTS. Then a macrostate of $\mathcal{M}$ is a triplet*

$$(s, c, e) \subseteq S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e).$$

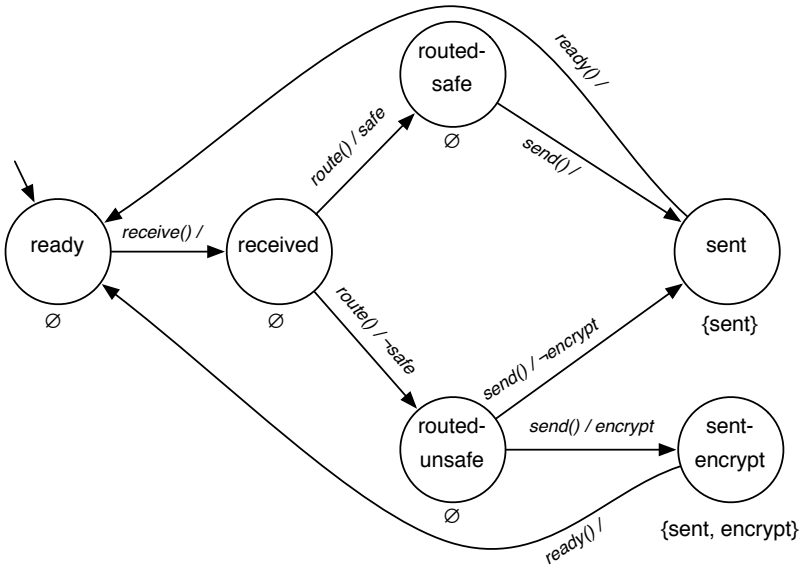

Fig. 1. The A-FTS modelling the adaptive routing protocol.

For example, one of the macrostates of the A-FTS presented in Example 8 is $(ready, \emptyset, \{safe\})$. If the A-FTS is in this macrostate, it means that the system is in state *ready*, has not the feature *encryption* enabled and executes in a $safe$ environment. Then once the action `receive()` is executed, the A-FTS can reach one of the following four macrostates: $(\texttt{receive()}, \emptyset, \emptyset)$, $(\texttt{receive()}, \emptyset, safe)$, $(\texttt{receive()}, encryption, \emptyset)$, and $(\texttt{receive()}, encryption, safe)$. The actual macrostate depends on how the environment evolves and how the system decides to reconfigure itself.

Definition 10 *A run in an A-FTS $\mathcal{M}$ is a sequence of the form*

$$(s_0, c_0, e_0)\alpha_0(s_1, c_1, e_1)\alpha_1 \ldots (s_i, c_i, e_i)\alpha_i \ldots$$

where $(s_0, c_0, e_0) \in I \times \mathcal{P}(F_s) \times \mathcal{P}(F_e)$ is called the initial *macrostate and where for every $i \in \mathbb{N}$ we have $\gamma(s_i, \alpha_i, s_{i+1})(c_i \cup e_i, c_{i+1} \cup e_{i+1})$.*

Note that this definition allows the system to start in any valid configuration. For instance, a run in the A-FTS described in Example 8 is

$$\begin{aligned}(ready, \emptyset, \{safe\})\ \texttt{receive()}\ (ready, \{encrypt\}, \emptyset)\ \texttt{route()}\\ (routed\text{-}unsafe, \{encrypt\}, \emptyset)\ \texttt{send()}\ (sent\text{-}encrypt)\ \texttt{ready()}\\ (ready, \emptyset, \{safe\}) \ldots \quad (3.6)\end{aligned}$$

As for FTS, we define the projection of an A-FTS onto a configuration c as the A-FTS obtained by setting its initial configuration to c. The resulting A-FTS is noted $\mathcal{M}_{|c}$ and is such that

$$[\![\mathcal{M}_{|c}]\!] = \{(s_0, c_0, e_0)\alpha_0(s_1, c_1, e_1)\alpha_1 \ldots (s_i, c_i, e_i)\alpha_i \cdots \in [\![\mathcal{M}]\!] \mid s_0 \in I \wedge c_0 = c \wedge \forall i \in \mathbb{N} \bullet (s_i, c_i, e_i) \subseteq S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e)\}. \quad (3.7)$$

Then the semantics of an A-FTS is a function that associates a system configuration c with the set of executions where the system starts in configuration c.

Definition 11 *Let $\mathcal{M}$ be an A-FTS. The semantics of $\mathcal{M}$ is the function*

$$[\![\mathcal{M}]\!] : \mathcal{P}(F_s) \to (S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e) \times Act)^\omega \bullet [\![\mathcal{M}]\!](c) = [\![\mathcal{M}_{|c}]\!] \quad (3.8)$$

According to the above semantics, there is a close relation between A-FTS, FTS and LTS. An FTS is an A-FTS where the environment has an established, unchanging configuration and where the system starts in any valid configuration and never modifies it, that is

$$\forall (s_0, c_0, e_0)\alpha_0(s_1, c_1, e_1)\alpha_1 \ldots (s_i, c_i, e_i)\alpha_i \cdots \in [\![\mathcal{M}]\!] \bullet \forall i \in N \bullet c_i = c_{i+1} \wedge e_i = e_{i+1}. \quad (3.9)$$

A sufficient condition for that condition to hold is that $\gamma(s, \alpha, s')(f, f')$ returns $\perp$ whenever $f \neq f'$. In this case, two runs differ only by (1) the actions chosen by the environment and (2) the states reached by the system. Furthermore, the projection of this FTS onto a configuration c (*i.e.* an LTS) is equivalent to the projection of this special A-FTS to c.

4 The AdaCTL Logic

To express properties that a DSPL must satisfy, we use a variant of the fCTL logic. The resulting logic, called *Adaptive Configuration Time Logic* (AdaCTL), extends the syntax of fCTL to allow further reasoning over the features. Also, its semantics is different because it takes into account that the system and the environment are not always allowed to change their own configuration. In this section, we introduce the syntax and the semantics of AdaCTL and provide an example of properties that can be expressed in this logic.

4.1 Syntax

We can classify the AdaCTL formulae into three categories. The first type of formula is called *feature* formula. It has the form

$$\Psi ::= [\chi]\Phi \quad (4.1)$$

where χ is a feature expression and Φ is a *state* formula. To increase readability, when the feature expression χ is equivalent to $\top$, we omit it; that is, for any state formula Φ, we define that

$$\Phi \triangleq [\top]\Phi. \tag{4.2}$$

A state formula is defined over a set AP of atomic propositions and is built according to the following grammar:

$$\Phi ::= \top \mid a \mid \Psi_1 \wedge \Psi_2 \mid \neg\Psi \mid \mathcal{A}\varphi \mid \mathcal{E}\varphi \tag{4.3}$$

where $a \in AP$, Ψ, Ψ_1 and Ψ_2 are feature formulae, and φ is a *path formula*. This latter category of AdaCTL formula is defined as follows:

$$\varphi ::= \bigcirc\Psi \mid \Psi_1 \mathrm{U}\ \Psi_2 \mid \Psi_1 \mathrm{R}\ \Psi_2 \tag{4.4}$$

where Ψ, Ψ_1, and Ψ_2 are feature formulae, $\bigcirc$ is the *next* operator, U is the *until* operator, and R is the *release* operator.

Before providing AdaCTL with a formal semantics, we first explain it intuitively for each type of formula. A feature formula $[\chi]\Phi$ means that if the system is in a given macrostate (s, c, e) such that the configuration of the system and the environment satisfy the feature expression χ (that is, $c \cup e \in [\![\chi]\!]$), then s must satisfy the state formula Φ. It is thus very similar to an fCTL formula. The difference is that in fCTL, a feature expression occurs before the top-level state formula only, whereas AdaCTL allows it to occur before any state formula. This provides more flexible ways to reason on the features. In particular, if we want to express that a feature f must be enabled, we may use the formula $[\neg f]\ \bot$, which is satisfied if and only if the system is in a configuration where f is enabled. Moreover, while authorizing feature expressions to occur at any level of a formula would not change the expressiveness of fCTL, it increases that of AdaCTL. For example, since the environment is modelled as a set of features varying over time, AdaCTL formulae can model changes of objectives with respect to what happened in the past.

A state formula is a formula defined over a state. Any macrostate satisfies the formula $\top$. A macrostate (s,c,e) satisfies a if and only if a belongs to the set of atomic propositions in $L(s)$; it satisfies the conjunction of two formulae if and only if it satisfies both. Also, the negation of a formula is satisfied if and only if the formula itself is not satisfied. The AdaCTL operator $\mathcal{E}$ is similar to the existential operator of CTL: a macrostate m satisfies the formula $\mathcal{E}\varphi$ if and only if there exists a path starting from m that satisfies φ.

The most subtle difference between AdaCTL and the other two logics lies in the semantics of formulae of the form $\mathcal{A}\varphi$. A macrostate m satisfies $\mathcal{A}\varphi$ if and only if starting from m, the system can ensure by means of reconfigurations that any forthcoming execution will satisfy φ regardless of the environment.

As in CTL, a path π satisfies the AdaCTL path formula $\bigcirc\Psi$ if and only if the first macrostate of π (that is, the macrostate following the initial one) satisfies the feature formula Ψ. A path π satisfies $\Psi_1 \mathrm{U} \Psi_2$ if and only if it eventually

reaches a macrostate m_j that satisfies Ψ_2 and every macrostate before m_j on π satisfies Ψ_1. Finally, π satisfies $\Psi_1 \mathrm{R} \Psi_2$ if and only if every macrostate reached by π satisfies Ψ_2 unless a previously reached macrostate satisfied Ψ_1.

From the until and the release operator, one can derive two additional, intensively used operators: *eventually* ($\Diamond$) and *forever* ($\Box$). Intuitively, a path π satisfies $\Diamond\Psi$ if and only if there exists a macrostate along π that satisfies Ψ; π satisfies $\Box\Psi$ if and only if every macrostate along this path satisfies Ψ. Formally, these two operators are obtained as follows:

$$[\chi]\mathcal{E}\Diamond\Psi = [\chi]\mathcal{E}(\top \text{ U } \Psi) \tag{4.5}$$

$$[\chi]\mathcal{A}\Diamond\Psi = [\chi]\mathcal{A}(\top \text{ U } \Psi) \tag{4.6}$$

$$[\chi]\mathcal{E}\Box\Psi = [\chi]\mathcal{E}(\top \text{ R } \Psi) \tag{4.7}$$

$$[\chi]\mathcal{A}\Box\Psi = [\chi]\mathcal{A}(\top \text{ R } \Psi) \tag{4.8}$$

Example 12 *We now provide an example of AdaCTL formula. Let us consider the A-FTS presented in Example 8 and the property according which the system must ensure that in an unsafe environment, no packet is sent before it is encrypted. We can express this property as the AdaCTL formula*

$$\mathcal{A}\Box([\neg safe]\mathcal{A}(\neg sent \text{ U}[\neg safe]encrypted)) \tag{4.9}$$

The $\Box$ operator is needed because the environment can become unsafe at any moment. According to this formula, from every macrostate where the environment feature $safe$ is disabled, the system must eventually reach a macrostate where either the atomic proposition `encrypted` *is satisfied or the environment is safe again; any macrostate reached in the mean time must be such that the atomic proposition* `sent` *is not satisfied.*

Example 13 *Assume we model the requirements for a satellite to normally always maintain altitude (a) and make observations (o), but in case the solar panels are damaged (failure d), the second requirement can be dropped:*

$$\mathcal{A}\Box(\ a \wedge [\neg d]o\ ) \tag{4.10}$$

In classical CTL, the R operator is derived from the operator of U and the negation. Consequently, $\Box$ is also derived from $\Diamond$ and the negation. However, as we will show further in this section, this definition is not suitable in *AdaCTL* because $\mathcal{A}$ and $\mathcal{E}$ are not dual. Hence the need for considering R as a primitive operator that cannot be derived from the others.

4.2 Semantics

Before providing AdaCTL with a formal semantics, we first formalise the notions of strategy for both the system and the environment. Intuitively, a strategy for the system determines how the systems reacts (that is, how it reconfigures itself) according to what happened in the past. We call that a *reconfiguration strategy.*

Definition 14 *Let $\mathcal{M} = (S, Act, trans, i, AP, L, d, \gamma)$ be an A-FTS. A* reconfiguration strategy *is a function*

$$Str_C : (S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e))^+ \times S \times \mathcal{P}(F_e) \to \mathcal{P}(F_s) \tag{4.11}$$

where X^* is the type of the sequences of Xs.

Intuitively, the system modifies its configuration according to the sequence of all the macrostates that have been previously visited, the next state that is reached and the next configuration of the environment.

Similarly, we can encode non-deterministic choices and uncontrolled configuration as a strategy for the environment.

Definition 15 *Let $\mathcal{M} = (S, Act, trans, i, AP, L, d, \gamma)$ be an A-FTS. An* environment strategy *is a function*

$$Str_E : (S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e))^+ \to Act \times \mathcal{P}(F_e) \tag{4.12}$$

An environment strategy thus associates the sequence of macrostates that have already been visited with an action and a new configuration for the environment. These definitions of strategy are closely related to those found in the *Alternating Time Logic* (ATL) theory [2]. If the notion of feature were absent, AdaCTL model checking could be regarded as a particular case of ATL model checking. We discuss the link between these two logics more thoroughly in Section 6.

Given an initial macrostate $init = (s_0, c_0, e_0)$, an environment strategy Str_E, and a reconfiguration strategy Str_C, applying Str_E and Str_C from $init$ results in a unique execution

$$Path(init, Str_C, Str_E) = (s_0, c_0, e_0)\alpha_0(s_1, c_1, e_1)\alpha_1 \dots$$

such that $\forall i \in \mathbb{N}$ we have

$$(\alpha_i, e_{i+1}) = Str_E(s_0, c_0, e_0, \dots, s_i, c_i, e_i) \tag{4.13}$$

$$c_{i+1} = Str_C(s_0, c_0, e_0, \dots, s_i, c_i, e_i, s_{i+1}, e_{i+1}). \tag{4.14}$$

This run is *valid* according to $\mathcal{M}$ if and only if it is part of the semantics of $\mathcal{M}$, that is, $Path(Init, Str_C, Str_E) \in [\![\mathcal{M}]\!]$. More generally, we denote by $Path(m, Str_C, Str_E)$ the path starting from a macrostate m induced by the environment strategy $Strat_E$ and the reconfiguration strategy $Strat_C$.

Following the definition of valid execution, we define that an environment strategy Str_E is *valid* according to $\mathcal{M}$ if and only if it cannot lead to invalid executions.

$$\forall init \in I \times \mathcal{P}(F_s) \times \mathcal{P}(F_e) \bullet \forall Str_C \bullet Path(init, Str_C, Str_E) \in [\![\mathcal{M}]\!] \tag{4.15}$$

We define similarly the validity of a reconfiguration strategy Str_C:

$$\forall init \in I \times \mathcal{P}(F_s) \times \mathcal{P}(F_e) \bullet \forall Str_E \bullet Path(init, Str_C, Str_E) \in [\![\mathcal{M}]\!] \tag{4.16}$$

From now on, we consider only valid strategies. Then, we define the semantics of AdaCTL as follows.

Definition 16 *Let $\mathcal{M}$ be an A-FTS, (s, c, e) one of its macrostates. Then the satisfiability of an AdaCTL feature or state formula by $\mathcal{M}$ in macrostate (s, c, e) is determined according to the following rules:*

$$
\begin{aligned}
\mathcal{M}, (s,c,e) \models\ & [\chi]\Phi && \Leftrightarrow c \cup e \notin [\![\chi]\!] \vee \mathcal{M}, (s,c,e) \models \Phi \\
\mathcal{M}, (s,c,e) \models\ & \top && \Leftrightarrow \top \\
\mathcal{M}, (s,c,e) \models\ & a && \Leftrightarrow a \in L(s) \\
\mathcal{M}, (s,c,e) \models\ & \Phi_1 \wedge \Phi_2 && \Leftrightarrow \mathcal{M}, (s,c,e) \models \Phi_1 \wedge \mathcal{M}, (s,c,e) \models \Phi_2 \\
\mathcal{M}, (s,c,e) \models\ & \neg\Phi && \Leftrightarrow \neg(\mathcal{M}, (s,c,e) \models \Phi) \\
\mathcal{M}, (s,c,e) \models\ & \mathcal{E}\varphi && \Leftrightarrow \exists Str_C \bullet \exists Str_E \bullet \mathcal{M}, Path((s,c,e), Str_C, Str_E) \models \varphi \\
\mathcal{M}, (s,c,e) \models\ & \mathcal{A}\varphi && \Leftrightarrow \exists Str_C \bullet \forall Str_E \bullet \mathcal{M}, Path((s,c,e), Str_C, Str_E) \models \varphi
\end{aligned}
$$

The semantics of path formulae is very similar to that of CTL path formulae:

$$
\begin{aligned}
&\mathcal{M}, \pi \models \ \bigcirc\Psi \ \Leftrightarrow \pi[1] \models \Psi \\
&\mathcal{M}, \pi \models \Psi_1 \mathrm{U} \Psi_2 \Leftrightarrow \exists j \geq 0 \bullet \pi[j] \models \Psi_2 \wedge \forall i \leq j \bullet \pi[i] \models \Psi_1 \\
&\mathcal{M}, \pi \models \Psi_1 \mathrm{R} \Psi_2 \Leftrightarrow (\forall j \geq 0 \bullet \pi[j] \models \Psi_2) \vee (\exists i \bullet \pi[i] \models \Psi_1 \wedge \forall k \leq i \bullet \pi[k] \models \Psi_2)
\end{aligned}
$$

where π is a path, $\pi[0]$ is the initial macrostate of π, and $\pi[i{+}1]$ is the macrostate following $\pi[i]$ in π.

According to the above, we define that $\mathcal{M}_{|c}$ satisfies an AdaCTL formula Ψ if and only if for any initial state i of $\mathcal{M}$ and environment configuration e, $\mathcal{M}$ satisfies the formula in macrostate (i, c, e); that is,

$$(\mathcal{M}_{|c} \models \Psi) \Leftrightarrow \forall i \in I \bullet \forall e \in \mathcal{P}(F_e) \bullet \mathcal{M}(i, c, e) \models \Psi. \tag{4.17}$$

This leads us to the more general satisfiability relation of an AdaCTL formula by an A-FTS. As for FTS and fCTL, this relation, noted $\models_F$ is not boolean [19]. Instead, it is defined as the set of configurations such that when starting in such a configuration, the A-FTS satisfies the formula.

Definition 17 *Let $\mathcal{M}$ be an A-FTS and Ψ an AdaCTL formula. Then,*

$$(\mathcal{M} \models_F \Psi) = \{c \in \mathcal{P}(F_s) \mid \mathcal{M}_{|c} \models \Psi\} \tag{4.18}$$

Definition 18 *A formula Ψ is called an* absolute *requirement if it contains no feature (i.e. no occurrence of the $[\chi]$ operator). It is called* conditional *if it contains non-adaptable system features. It is called* adaptive *if it contains the $\mathcal{A}$ operator. A system is called* adaptive *if it has adaptive requirements and adaptable features.*

Definition 19 *An adaptive system $\mathcal{M}$ with requirements Φ is called* resilient *if there is an initial configuration such that all its requirements are satisfied: $\mathcal{M} \models_F \Phi \neq \emptyset$.*

This implies that, for each adaptive requirement, the system must be equipped with adaptation strategies that allow him to react to any environment (re)configuration, in particular to any failure.

For instance, if $\mathcal{M}$ is the A-FTS presented in Example 8 and Ψ is the adaptive requirement given in Example 12, we have

$$(\mathcal{M} \models_F \Psi) = \mathcal{P}(F_s) \tag{4.19}$$

since for any initial configuration, there exists a reconfiguration strategy that ensures the satisfaction of Ψ. One such strategy would be to enable the feature *encrypt* as soon as the system reaches the state `received`. However, if setting up this feature requires some time (for downloading the encryption code, etc.), the system has to wait that the feature is enabled before sending the message.

Note that according to the above semantics, $\mathcal{A}$ and $\mathcal{E}$ are not dual. We prove this for the $\bigcirc$ operator.

Theorem 20 *Let Ψ be an AdaCTL feature formula. We have*

$$\mathcal{A} \bigcirc \Psi \neq \neg(\mathcal{E} \bigcirc \neg\Psi) \tag{4.20}$$

Proof. Let us assume that $\mathcal{A}$ and $\mathcal{E}$ are dual for the $\bigcirc$ operator. Let us consider the two AdaCTL formulae $\mathcal{A} \bigcirc \mathcal{A} \bigcirc a$ and $\mathcal{E} \bigcirc \mathcal{E} \bigcirc \neg a$ where a is an atomic proposition. Then, for any A-FTS $\mathcal{M}$ and system configuration c, if $\mathcal{M}_{|c}$ satisfies the former then it does not satisfy the latter and vice-versa. Let $\mathcal{M}$ be the A-FTS shown in Figure 2 where f is a feature of the system. It turns out that $\mathcal{M}_{|c} \models \mathcal{A} \bigcirc \mathcal{A} \bigcirc a$ for any configuration c. Indeed, once in state 2 the system can change its configuration such that f is not enabled. In this case, only state 3 is reachable and the formula is thus satisfied regardless of the action chosen by the environment and its configuration. On the other hand, $\mathcal{M}_{|c} \models \mathcal{E} \bigcirc \mathcal{E} \bigcirc \neg a$ for any c as well. Once in state 2, if the system change its configuration such that f is enabled, the system can reach state 4. Since $\mathcal{M}$ satisfies both formulae and given that it is impossible for an A-FTS to satisfy both a formula and its negation, the duality law does not hold. ■

The above counterexample also denies the duality law for the $\Diamond$, the $\Box$, the Uand the Roperators. Since the proof is very similar, we omit it.

Theorem 21 *Let Ψ be an AdaCTL feature formula. We have*

$$\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2) \neq \neg(\mathcal{E}(\neg\Psi_1 \mathrm{R} \neg\Psi_2)) \tag{4.21}$$

$$\mathcal{E}(\Psi_1 \mathrm{U} \Psi_2) \neq \neg(\mathcal{A}(\neg\Psi_1 \mathrm{R} \neg\Psi_2)) \tag{4.22}$$

$$\mathcal{A}\Diamond\Psi \neq \neg(\mathcal{E}\Box\neg\Psi) \tag{4.23}$$

$$\mathcal{E}\Diamond\Psi \neq \neg(\mathcal{A}\Box\neg\Psi). \tag{4.24}$$

This implies that $\mathcal{A}$ cannot be expressed in terms of $\mathcal{E}$. Thus, when model checking an A-FTS against an AdaCTL formula, we have to consider the operator $\mathcal{A}$ and $\mathcal{E}$ separately. On the contrary, the usual *expansion laws* still hold. Basically,

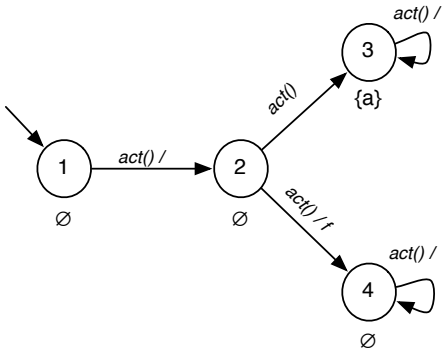

Fig. 2. Counterexample for the duality laws.

an expansion law allows to express an operator in terms of formulae that the current state must satisfy to satisfy the formula defined by the operator. For instance, the expansion law for $\mathcal{A}(\Psi_1 \mathrm{U} \Psi 2)$ expresses that this formula is satisfied if and only if Ψ_2 immediately holds or if Ψ_1 holds and the formula holds in the next state. As for CTL, the AdaCTL model checking algorithms make intensively use of these laws, as we will see in the next section.

Theorem 22 *Let Ψ, Ψ_1, Ψ_2 be AdaCTL formulae. Then we have the following expansion laws:*

$$\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2) \equiv \Psi_2 \vee (\Psi_1 \wedge \mathcal{A} \bigcirc \mathcal{A}(\Psi_1 \mathrm{U} \Psi_2)) \tag{4.25}$$
$$\mathcal{A}(\Psi_1 \mathrm{R} \Psi_2) \equiv \Psi_2 \wedge (\Psi_1 \vee \mathcal{A} \bigcirc \mathcal{A}(\Psi_1 \mathrm{R} \Psi_2)) \tag{4.26}$$
$$\mathcal{A}(\Diamond \Psi) \equiv \Psi \wedge \mathcal{A} \bigcirc \mathcal{A} \Diamond \Psi \tag{4.27}$$
$$\mathcal{A}(\Box \Psi) \equiv \Psi \wedge \mathcal{A} \bigcirc \mathcal{A} \Box \Psi \tag{4.28}$$
$$\mathcal{E}(\Psi_1 \mathrm{U} \Psi_2) \equiv \Psi_2 \vee (\Psi_1 \wedge \mathcal{E} \bigcirc \mathcal{E}(\Psi_1 \mathrm{U} \Psi_2)) \tag{4.29}$$
$$\mathcal{E}(\Psi_1 \mathrm{R} \Psi_2) \equiv \Psi_2 \wedge (\Psi_1 \vee \mathcal{E} \bigcirc \mathcal{E}(\Psi_1 \mathrm{R} \Psi_2)) \tag{4.30}$$
$$\mathcal{E}(\Diamond \Psi) \equiv \Psi \wedge \mathcal{E} \bigcirc \mathcal{E} \Diamond \Psi \tag{4.31}$$
$$\mathcal{E}(\Box \Psi) \equiv \Psi \wedge \mathcal{E} \bigcirc \mathcal{E} \Box \Psi \tag{4.32}$$

Proof. We prove only the expansion law of $\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2)$. The other proofs involving the $\mathcal{A}$ operator either can be derived from this one or follow a similar pattern. The proofs related to $\mathcal{E}$ are implied from the expansion laws in CTL.

For any $m = (s, c, e)$, St_E, and St_C, the initial state of $Path(m, St_C, St_E)$ is m and thus depends on neither St_C nor St_E. Accordingly, the semantics of $\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2)$ is equivalent to

$$(\mathcal{M}, m \models \Psi_2) \vee ((\mathcal{M}, m \models \Psi_1) \wedge \exists Str_C \bullet \forall Str_E \bullet \\ Path(Path(m, Str_C, Str_E)[1], Str_C, Str_E) \models \Psi_1 \mathrm{U} \Psi_2 \tag{4.33}$$

On the other hand,

$$\mathcal{M}, m \models (\Psi_2 \vee (\Psi_1 \wedge \mathcal{A} \bigcirc \mathcal{A}(\Psi_1 \mathrm{U} \Psi 2))) \tag{4.34}$$

is equivalent to

$$(\mathcal{M}, m \models \Psi_2) \vee ((\mathcal{M}, m \models \Psi_1) \wedge \exists Str'_C \bullet \forall Str'_E \bullet \\ Path(m, Str'_C, Str'_E) \models \bigcirc \mathcal{A}(\Psi_1 \mathrm{U} \Psi_2) \tag{4.35}$$

which can be re-written as

$$(\mathcal{M}, m \models \Psi_2) \vee ((\mathcal{M}, m \models \Psi_1) \wedge \exists Str'_C \bullet \forall Str'_E \bullet \exists Str''_C \bullet \forall Str''_E \bullet \\ Path(Path(m, Str'_C, Str'_E)[1], Str''_C, Str''_E) \models \Psi_1 \mathrm{U} \Psi_2 \tag{4.36}$$

We immediately have that Equation (4.33) implies Equation (4.36); in this case, we have $Str_C = Str'_C = Str''_C$. Next, we define the strategy Str'''_C such that Str'''_C behaves like Str'_C for the first transition, and like Str''_C for the subsequent ones. Then for any Str'''_E we have that

$$Path(Path(m, Str'''_C, Str'''_E)[1], Str'''_C, Str'''_E) \models \Psi_1 \mathrm{U} \Psi_2. \tag{4.37}$$

Equations (4.33) and (4.36) are thus equivalent and we have proven the expansion law. ■

The expansion laws for $\mathcal{A}(\varphi)$ imply that if Str_C is the reconfiguration strategy such that $\forall Str_E \bullet Path(m, Str_C, Str_E) \models \varphi$ then every macrostate m' reached on this path is such that $\mathcal{M}, m' \models \mathcal{A}(\varphi)$.

5 Algorithms

As previously mentioned, model checking an A-FTS against an AdaCTL formula comes down to identifying the initial configurations such that for any initial state and initial environment configuration, the A-FTS satisfies the formula when the system starts in such a configuration. In this section, we propose an algorithm to compute the satisfaction relation between an A-FTS $\mathcal{M}$ and an AdaCTL formula Ψ. For the sake of readability and conciseness, we present only the algorithms in their explicit form. Following our previous work on fCTL model checking [16], we can transform them into symbolic algorithms by using the same encoding techniques. The basic idea is to encode states, configurations, and the transition relation as Boolean functions. Then, symbolic algorithms work with those functions instead of their explicit counterpart (see [16] for more details). Note that the conversion of the transition relation in A-FTS is facilitated since it is already defined as a Boolean function.

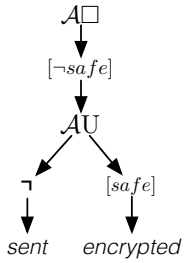

Fig. 3. Parse-tree of the formula $\mathcal{A}(\Box[\neg safe]\mathcal{A}(\neg sent \text{ U } [\neg safe]encrypted))$.

5.1 AdaCTL Model Checking

We first decompose Ψ into its *parse tree*. Basically, the parse tree of a formula is a tree such that each node represents a subformula of Ψ, the root is Ψ itself, and the leaves are atomic propositions. For example, the parse tree of the AdaCTL formula given in Example 12 is shown in Figure 3. Then, starting from the leaves and for every subformula Ψ', we compute the set of macrostates

$$Sat(\Psi') = \{(s, c, e) \in S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e) \mid \mathcal{M}, (s, c, e) \models \Psi'\} \tag{5.1}$$

for every subformula Ψ' of Ψ. Once we have determined the set of macrostates satisfying the whole formula Ψ, we infer the set $(\mathcal{M} \models_F \Psi)$. This method allows us to decompose the verification of a formula into smaller and independent problems. It is thus similar to the fCTL and the standard CTL model checking algorithms [16, 15]. However, it has to cope with (1) macrostates instead of states, (2) dynamic features, and (3) the $\mathcal{A}$ quantifier, which does not exist in the other two logics.

The computation of the set of macrostates satisfying feature and state formulae directly follows from their semantics.

$$Sat([\chi]\Phi) = \{(s, c, e) \in S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e) \mid c \cup e \notin [\![\chi]\!]\} \cup Sat(\Phi) \tag{5.2}$$

$$Sat(\top) = S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e) \tag{5.3}$$

$$Sat(a) = \{(s, c, e) \in S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e) \mid a \in L(s)\} \tag{5.4}$$

$$Sat(\Psi_1 \wedge \Psi_2) = Sat(\Psi_1) \cap Sat(\Psi_2) \tag{5.5}$$

$$Sat(\neg\Psi) = S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e) \setminus Sat(\Psi) \tag{5.6}$$

The first step towards computing a set of the form $Sat(\mathcal{E}\varphi)$ or $Sat(\mathcal{A}\varphi)$ is the definition and the computation of predecessors set. The notion of predecessors is, however, different depending on whether we consider the $\mathcal{E}$ quantifier or the $\mathcal{A}$ quantifier.

In the former case, a macrostate m is an $\mathcal{E}$-predecessor of m' if and only if from m, the macrostate m' can be reached in one transition. More generally, let

$\mathcal{S}$ be a set of macrostates. Then the $\mathcal{E}$-predecessors set of $\mathcal{S}$, noted $Pre_{\mathcal{E}}(\mathcal{S})$, is defined as the set of macrostates from which a macrostate in $\mathcal{S}$ can be reached in one transition. Formally,

$$Pre_{\mathcal{E}}(\mathcal{S}) = \{(s,c,e) \mid \exists \alpha \in Act, (s',c',e') \in \mathcal{S} \bullet \gamma(s,\alpha,s')(c \cup e, c' \cup e')\}\}. \quad (5.7)$$

In the latter case, we define that m is an $\mathcal{A}$-predecessor of m' if the system can come up with a valid strategy such that when in m, it is ensured that m' will be reached after the execution of the next transition. Similarly to the previous case, we define the $\mathcal{A}$-predecessor set of a set of macrostates $\mathcal{S}$. Intuitively, it is the set of macrostates such that the system can ensure that after the execution of the next transition, it will reach a macrostate of $\mathcal{S}$. Given that the system chooses its next configuration after the next state and the new environment configuration have been determined, it is defined as

$$\begin{aligned} Pre_{\mathcal{A}}(\mathcal{S}) = \{(s,c,e) \mid \forall \alpha \in Act, s' \in S, e' \in \mathcal{P}(F_e) \bullet \\ (\exists c'' \in \mathcal{P}(F_s) \bullet \gamma(s,\alpha,s')(c \cup e, c'' \cup e') \Rightarrow \\ (\exists c' \in \mathcal{P}(F_s) \bullet \gamma(s,\alpha,s')(c \cup e, c' \cup e') \wedge (s',c',e') \in \mathcal{S})). \end{aligned} \quad (5.8)$$

Let

$$\begin{aligned} D(s,c,e,\alpha,s',e') = \{(s',c',e') \in S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e) \\ \mid \gamma(s,\alpha,s')(c \cup e, c' \cup e')\} \end{aligned} \quad (5.9)$$

then $Pre_{\mathcal{A}}(\mathcal{S})$ is equivalent to

$$\bigcap_{\alpha,s',e'} \{(s,c,e) \mid (D(s,c,e,\alpha,s',e') = \emptyset) \vee (D(s,c,e,\alpha,s',e') \cap \mathcal{S} \neq \emptyset)\} \quad (5.10)$$

Since the semantics of $\mathcal{E}$ is similar to that of the existential quantifier in CTL, the set of macrostates satisfying a formula of the form $\mathcal{E}\varphi$ is computed as in the basic CTL model checking algorithm. $Sat(\mathcal{E} \bigcirc \Psi)$ is the set of macrostate such that in one transition, the system can reach a state s', be in configuration c' and the environment can be in a configuration e' such that (s',c',e') satisfies φ. Formally, we have

$$Sat(\mathcal{E}(\bigcirc \Psi)) = Pre_{\mathcal{E}}(Sat(\Psi)) \quad (5.11)$$

On the other hand, $\mathcal{E}(\Psi_1 \mathrm{U} \Psi_2)$ is the smallest set $\mathcal{S}$ satisfying

$$Sat(\Psi_2) \subseteq \mathcal{S} \quad (5.12)$$

$$Sat(\Psi_1) \cap Pre_{\mathcal{E}}(S) \subseteq \mathcal{S}; \quad (5.13)$$

$\mathcal{E}(\Psi_1 \mathrm{R} \Psi_2)$ is the largest set $\mathcal{S}$ satisfying

$$\mathcal{S} \subseteq Sat(\Psi_2) \quad (5.14)$$

$$\mathcal{S} \subseteq Sat(\Psi_1) \cup Pre_{\mathcal{E}}(\mathcal{S}) \quad (5.15)$$

Both can be computed through fixed-point computation [15]. The corresponding algorithms being identical to their CTL counterpart, we omit them here.

We focus now on the $\mathcal{A}$ quantifier. Basically, the satisfaction sets have a similar definition than those for the $\mathcal{E}$ quantifier; the difference is that the $\mathcal{A}$ quantifier requires the computation of the $\mathcal{A}$-predecessors instead of the $\mathcal{E}$-predecessors. For the next operator, we have the following.

Theorem 23

$$Sat(\mathcal{A} \bigcirc \Psi) = Pre_{\mathcal{A}}(\Psi) \tag{5.16}$$

Proof. Follows from the semantics of $\mathcal{A} \bigcirc \Psi$ *and the definition of* $Pre_{\mathcal{A}}(\Psi)$.

We obtain $Sat(\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2))$ through a fixed-point computation. This result relies on the following theorem.

Theorem 24 $Sat(\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2))$ *is the smallest set* $\mathcal{S}$ *satisfying*

$$Sat(\Psi_2) \subseteq \mathcal{S} \tag{5.17}$$

$$Sat(\Psi_1) \cap Pre_{\mathcal{A}}(\mathcal{S}) \subseteq \mathcal{S} \tag{5.18}$$

Proof. First, it directly follows from the expansion law of the U*operator (see Theorem 22) that* $Sat(\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2))$ *satisfies Equations (5.17–5.18). It remains to show that any set* $\mathcal{S}$ *satisfying those Equations is a superset of* $Sat(\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2))$.

Let $m \in Sat(\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2))$. *We distinguish between* $m \in Sat(\Phi_2)$ *and* $m \notin Sat(\Phi_2)$.

(a) If $m \in Sat(\Phi_2)$ *then* $m \in \mathcal{S}$ *by Equation 5.17.*
(b) Otherwise, by definition of $Sat(\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2))$, *we have*

$$\exists Str_C \bullet \forall Str_E \bullet Path(m, Str_C, Str_E) \models \Psi_1 \mathrm{U} \Psi_2.$$

It means that Str_C *ensures that from* m, *we eventually reach a macrostate that is in* $Sat(\Psi_2)$. *Let* k *be the largest number of transitions needed by* Str_C *to reach from* m *such a macrostate, and let* m_k *this macrostate. Then* $m_k \in \mathcal{S}$ *by Equation 5.17. Before reaching* m_k, *we must first reach a macrostate* $m_{k-1} \in Sat(\Psi_1)$ *such that from* m_{k-1}, Str_C *ensures to reach* m_k *in one transition regardless of the strategy of the environment. Such a macrostate is reached in at most* $k-1$ *transitions, and belongs to* $Pre_{\mathcal{A}}(\{m_k\}) \subseteq Pre_{\mathcal{A}}(Sat(\Psi_2))$, *and is thus in* $\mathcal{S}$ *by Equation (5.18). By induction on the maximum number of transitions needed to reach a macrostate in* $Sat(\Psi_2)$, *we obtain that* $m \in \mathcal{S}$.

The proof is then complete. ■

In order to compute $Sat(\Psi_1 \mathrm{U} \Psi_2)$, we define the function

$$T_{\mathrm{U}} : \mathcal{P}(S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e)) \to \mathcal{P}(S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e)) \\ \bullet\ T_{\mathrm{U}}(\mathcal{S}) = \mathcal{S} \cup (Pre_{\mathcal{A}}(\mathcal{S}) \cap Sat(\Phi_1)). \tag{5.19}$$

Then, according to the Knaster-Tarski theorem and following Theorem 24, this set is the fixed-point of the function T_{U} when it is first applied to $Sat(\Psi_2)$, that is,

$$Sat(\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2)) = \mathcal{S}_i \bullet \forall j \geq i \bullet \mathcal{S}_j = \mathcal{S}_i \tag{5.20}$$

where $\forall j \in \mathbb{N}$

$$\mathcal{S}_0 = Sat(\Psi_2) \tag{5.21}$$

$$\mathcal{S}_{j+1} = T(\mathcal{S}_j) \tag{5.22}$$

The computation of $Sat(\mathcal{A}(\Psi_1 \mathrm{R} \Psi_2))$ follows a very similar procedure. For this reason, we omit the proof.

Theorem 25 *$Sat(\mathcal{A}(\Psi_1 \mathrm{R} \Psi_2))$ is the largest set $\mathcal{S}$ satisfying*

$$\mathcal{S} \subseteq Sat(\Psi_2) \tag{5.23}$$

$$\mathcal{S} \subseteq Sat(\Psi_1) \cup Pre_{\mathcal{A}}(\mathcal{S}) \tag{5.24}$$

Accordingly, this set can be computed as the fixed-point of the function

$$T_{\mathrm{R}} : \mathcal{P}(S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e)) \to \mathcal{P}(S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e)) \bullet T_{\mathrm{R}}(\mathcal{S}) = \mathcal{S} \cap Pre_{\mathcal{A}}(\mathcal{S}). \tag{5.25}$$

first applied to $Sat(\Psi_2)$.

Example 26 *We illustrate the definitions of $\mathcal{E}$-predecessors and $\mathcal{A}$-predecessors, as well as the above model checking algorithm. Let us consider the small A-FTS shown in Figure 2, which we verify against the formulae $\mathcal{E} \bigcirc \mathcal{E} \bigcirc \neg a$ and $\mathcal{A} \bigcirc \mathcal{A} \bigcirc a$. First, the algorithm determines which macrostates satisfy the atomic proposition a:*

$$Sat(a) = \{(3, c, e)\}.$$

As $\neg a$ occurs in the first formulae, it computes the corresponding satisfaction set by complementing the above:

$$Sat(\neg a) = \{(i, c, e) \mid i \in \{1, 2, 4\}\}.$$

We focus on $\mathcal{E}$-predecessors first. A macrostate satisfies $\mathcal{E} \bigcirc \neg a$ if and only if from this macrostate, the system may reach a set in $Sat(\neg a)$ in one transition. Hence,

$$Sat(\mathcal{E} \bigcirc \neg a) = \{(i, c, e) \mid i \in \{1, 4\} \vee (i = 2 \wedge f \in c)\}.$$

Indeed, from state 1 the system can reach state 2 regardless of its configuration and that of the environment; from state 2 it can reach state 4 if feature f is enabled; and it can always loop on state 4. For the same reasons, the macrostates satisfying the whole property is given by

$$Sat(\mathcal{E} \bigcirc \mathcal{E} \bigcirc \neg a) = \{(i, c, e) \mid i \in \{1, 4\} \vee (i = 2 \wedge f \in c)\}.$$

A macrostate satisfies $\mathcal{A} \bigcirc a$ if and only if from this macrostate, the system can ensure it will reach a macrostate satisfying a. Regardless of the configuration of the system and the environment, the former will remain in state 3 once it reaches this state. Moreover, if the system is in state 2 and feature f is disabled, it will necessarily reach state 3 after the next transition. From state 1, the system cannot reach state 3 in one transition. The corresponding satisfaction set is thus:

$$Sat(\mathcal{A} \bigcirc a) = \{(3, c, e)\} \cup \{(2, c, e) \mid f \notin c\}$$

The definition of $Sat(\mathcal{A} \bigcirc \mathcal{A} \bigcirc a)$ is identical except that this time, the system can satisfy the formula from state 1. The configuration of the system does not matter here since the system may modify it once it reaches state 2. We have

$$Sat(\mathcal{A} \bigcirc \mathcal{A} \bigcirc a) = \{(3, c, e)\} \cup \{(2, c, e) \mid f \notin c\} \cup \{1, c, e\}.$$

5.2 Time Complexity

We now discuss the computational time complexity of checking an A-FTS $\mathcal{M}$ against an arbitrary AdaCTL formula Ψ. Our algorithm recursively computes the satisfactions sets of the subformulae of Ψ. Its time complexity is thus linear in the size of Ψ. The satisfaction sets that are the most costly to compute are those for the U and the R operators. Let us assume that we encode the satisfaction sets and the transition relation symbolically. As in classical CTL model checking, the time complexity of computing $Sat(\mathcal{E}(\Phi_1 \mathrm{U} \Phi_2))$ and $Sat(\mathcal{E}(\Phi_2 \mathrm{R} \Phi_2))$ is bounded by the number of macrostates, i.e., $|S|.2^{|F_s|}.2^{|F_e|}$. This is because when computing the corresponding smallest (resp. greatest) fixed-point, if the fixed-point has not been reached then the application of the corresponding function removes (resp. adds) at least one element. Computing the sets $Sat(\mathcal{A}(\Phi_1 \mathrm{U} \Phi_2))$ and $Sat(\mathcal{A}(\Psi_1 \mathrm{R} \Phi_2))$ is more costly because each application of the functions T_{U} and T_{R} requires the computation of $Pre_{\mathcal{A}}(\mathcal{S})$. The time complexity of this computation is bounded by the number of states multiplied the number of environment configurations, that is, $|S|.2^{|F_e|}$. Since the number of needed applications of T_{U} and T_{R} is also bounded by the number of macrostates, we obtain the following result.

Theorem 27 *The time complexity of model checking an A-FTS against an AdaCTL formula Ψ is bounded by $\mathcal{O}(|S|^2.2^{2.|F_e|}.2^{|F_s|})$.*

Although it is theoretically dominated by $2^{2.|F_e|}$ and is thus in EXPTIME, in practice $|S|^2$ is often bigger.

6 Related Work

To the best of our knowledge, this paper is the first to tackle formal verification of self-adaptive SPLs. Consequently, we can only discuss related work in static SPL model checking and verification of adaptive systems.

6.1 SPL Model Checking

The need for quality assurance techniques in SPLs has been recognized as an important issue and we have observed the emergence of several techniques for solving the SPL model-checking problem. Most of them rely on the use of an automata-based formalism to model the behaviour of an SPL, and on the definition of dedicated checking algorithms. Fischbein *et al.* [30] were the first to propose modal transition systems to model product lines. In a nutshell, modal transition systems are LTS with mandatory and optional transitions, the latter being transitions that are not executable by all the products. To make this formalism more suitable in the context of SPLs, Fantechi and Gnesi [26] enriched it with variability operators and Asirelli *et al.* [6] equipped it with a logic able to express constraints on variable behaviour. Similarly, Gruler *et al.* [36] introduced PL-CCS, an extension of the CCS process algebra with variability operators able to express optionality. Instead of introducing a new formalism, Li *et al.* [40] proposed to model the behaviour of features with independent, single-system models. More precisely, they model both the system without features and the features as finite state machines. Then, the behaviour of a specific product is obtained by clinging its features onto the system.

As explained by Apel *et al.* [4], one can also use single-system model-checking to verify an SPL. In this case, however, the model-checker determines only whether or not there exists a product violating an intended property. The advantage of this approach is that it allows one to benefit from all the existing optimisations implemented in classical model-checkers. However, the goal of our work is more general since we want to pinpoint exactly *all* the misbehaving products or configurations.

The closest work on SPLs verification related to this paper is our previous work about FTS. In [17], we introduced a first definition of FTS, in which transitions are labelled with features and can have priority over each other. We also designed an algorithm for model checking an FTS against an LTL formula. In [16], we provided a new definition of FTS based on feature expressions (*i.e.* the one given in this paper); we also defined the fCTL logic and proposed symbolic model-checking algorithms for FTS. In a recent work [19], we extended the notion of simulation to the context of SPLs and we showed how FTS abstraction can reduce the verification time. In another work, we made use of this new relation to identify special classes of features and reduced the overhead of reverification when such features are introduced in an SPL [18].

The major difference between FTS and A-FTS are (1) the presence of a (possibly hostile) environment, (2) the presence of variability in both the system and its environment, and (3) the ability of the system and the environment to change their features at runtime. Because of those differences, reasoning on A-FTS requires the definition of a new logic to define the properties to be verified, *i.e.*, AdaCTL. In particular, fCTL is not suitable in this context because it permits to reason on neither dynamic features nor an external environment. AdaCTL is thus different from fCTL in both syntax and semantics, in the same

way as the classical logics ATL and CTL are different. A comparison between AdaCTL and ATL is given further in this section.

6.2 Verification of Adaptive Systems

The verification of adaptive systems is a topic that received a lot of attention from the scientific community. The work related to that context being particularly large, we focus here on the most recent results.

In their research roadmap for adaptive systems [13], Cheng *et al.* stated that in the context of adaptive systems, the objective of quality assurance is to provide evidence that the system is able to cope with changes in its objectives and its environment. The classical validation and verification methods being meant for stable systems, there is a crucial need for novel techniques specific to adaptive systems. They presented a framework for adaptive systems assurance, in which the system, the goals, and the context are subject to modifications. This results in a succession of models for the system and properties to verify. In our work, such a model is the LTS resulting from fixing the configuration of the state and the environment and removing the transitions unavailable for these configurations; the succession of properties can be expressed by AdaCTL feature formula.

Following the idea of representing adaptive systems as a succession of models, several verification methods model them as a set of programs [1, 38, 39, 51]. To ensure the satisfaction of intended properties in an unstable environment, the system is able to make transitions between those programs. The execution of such transitions is called an adaptation. By performing those, the system modifies its future behaviour. In this context, one distinguishes between local properties that specific programs must satisfy, global properties that must be satisfied by any execution of the system, and transitional properties that must hold during an adaptation.

To specify the transitional properties, Zhang *et al.* proposed a new logic called A-LTL [50]. In their recent work [51], they provided an algorithm based on marking to verify an adaptive system against an A-LTL formula. Although we do not tackle the same problems, there are similarities in their work and ours. Transposed to our work, a program of an adaptive system can be regarded as a particular configuration. Although A-LTL and AdaCTL have incomparable expressiveness, we can also express properties specific to some configurations as well as transitional properties by using AdaCTL feature formulae.

Closer to the notion of dynamic software product lines, Kulkarni *et al.* [39] consider that an adaptive system is a program able to add or remove components during runtime. Our definition of A-FTS is more general, as the effect of features is not limited to the addition or the removal of components. Instead of traditional model checking, they use proof lattice as an alternative solution for verifying that all possible adaptations satisfy all the global properties.

Instead of functional requirements, Filieri *et al.* [27] tackles the verification of non-functional requirements like reliability in the context of adaptive systems.

For this purpose, they propose novel algorithms for checking efficiently parametric Markov models (*viz.* parametric discrete-time Markov chains). Combining our work with theirs is an interesting perspective, as it could allow us to quantify the impact of adding or removing features at runtime in terms of reliability, performance or even energy consumption.

6.3 AdaCTL and ATL

As briefly mentioned in Section 4, there is a close relationship between AdaCTL and the *Alternating Time Logic* (ATL) of Alur *et al.* [2]. Without going into much detail or providing formal proofs, we compare our work with theirs and identify the commonalities and differences between the two. ATL is a logic able to express temporal properties on multi-agent systems and concurrent game structures. It provides a special quantifier $<< A >>$ where A is a set of agents or players working together to meet specific goals. The semantics of this operator makes use of a definition of strategy as well : $<< A >> \varphi$ is satisfied if and only if the players in A can find a strategy such that any execution following this strategy satisfies φ. Given that definition, the AdaCTL quantifier $\mathcal{A}$ is clearly similar to $<< Sys >>$ whereas $\mathcal{E}$ is similar to $<< Sys, Env >>$, Sys being the system and Env being the environment.

The difference between the two logics is that in AdaCTL, the transition relation depends on the configuration of both the adaptable and the non-adaptable features. Because of these features, the satisfiability relation is not binary and is thus more general than in ATL. Similarly, an A-FTS with a unique initial configuration can be translated into a two-player turn-based concurrent game structure. In this game, the players are the system and the environment. The action of the former is the choice of its new configuration. For the latter, it is the choice of an action and of its new configuration. In the end, this work can be regarded as a generalization of turn-based, 2-players concurrent game structures in the same way that FTS generalizes LTS.

7 Conclusion

We have presented a well-founded framework for the modelling and analysis of (self-)adaptive systems. We proposed a fundamental model, A-FTS, and a logic, AdaCTL, that are the basis for algorithms for analyzing resilience. This brings a number of benefits:

1. A sound theoretical basis;
2. An integration of static adaptation and dynamic adaptation, in its two variants: external adaptation and self-adaption by applying a pre-programmed change at the adequate point of time;
3. A clear, checkable definition of resilience;
4. Providing counterexamples when resilience fails.

These benefits mainly impact the predictability of self-adaptive systems.

However, we are well aware that many topics need further development before our methodology can be used routinely by engineers. A-FTS are just a fundamental model, that is used by the tools but leads to lengthy descriptions, difficult to manage by humans. It is therefore important to provide more manageable *high-level languages*, that can be compiled (on-the-fly) to A-FTS. These languages need to cover different *levels of abstraction*: the most abstract levels allowing a rapid analysis, while the most detailed can be directly used to generate executable code. This raise the question of *refinement*: can we guarantee that the analysis performed at the abstract level remains valid at the more concrete level? This question can be solved e.g. using alternating refinement [3]. This refinement should be *compositional*, so that modules of the system can be detailed independently, allowing teams to operate in parallel, on one hand, and analysis tools to cope with complexity, on the other.

Some features have a cross-cutting nature: we cannot simply add a module to the system to realize them. That is why we designed A-FTS with a low grain, where individual transitions can refer to features. As a consequence, first, features are spread over the whole A-FTS, and may be difficult to grasp; second, the addition of a supplementary feature is difficult, since each transition might need a revision. We plan thus to use (extensions of) the *aspect-oriented* approach to maintain a more localized and independent description of each feature (previous work on this topic includes [16, 31, 41]). The addition of a new feature will then hopefully be more understandable. We consider as still unrealistic, though, to hope that all features can be developed without being aware of other features, and that the weaving process will solve all emergent interactions. This raises the problem of defining, detecting and helping to solve *feature interactions*. A number of interesting approaches [34, 10] are available, but the problem is still pressing and largely unsolved.

We modelled failures as a subtype of environment features, which allows us to describe failure modes and effects. To have a realistic analysis of failures, we need to also model their *probabilities*, and to integrate (at least) classical methods for failure and reliability analysis [25].

A-FTS is based on the notion of a global state, so that all the information is available to both the system and its environment, and the strategies computed can rely on the unbounded past. In our setting, it is demonstrated that a bounded amount of information about the past is sufficient (finite memory). Techniques such as antichains [28] can be used to heuristically compute strategies that require a small memory. However, assuming complete observation is not realistic in most systems, and in general we need to switch to the notion of *partial observation* [14, 5]. Unfortunately, the problem becomes highly complex and often even undecidable [11]. The known (expensive) techniques rely on building all possible evolutions of the environment corresponding to the observations so far [14, 37]. A particular, easier problem is the *diagnostic* of failures [46]: from the partial observation provided by its sensors, can the system infer what has failed [46, 9]? Can it perform diagnostic actions to help inferring it?

Can it remedy to possible failures, even with a partial, ongoing diagnostic [44]? Can it perform the diagnostic against an active adversary [9]?

On the other hand, we have assumed that the self-adaptive system can choose among a set of preprogrammed dynamic system features. This more powerful than it seems, since the system can, if needed, chain several features to construct a complex plan to resist to a hostile environment. In particular, we can model *self-programming* systems by giving as features, the atomic instructions to be assembled. If the atomic instructions are infinite in number, however, our technique does not apply. In the long term, this might become a relevant challenge for deeply self-adaptive autonomous systems.

Since they are domain-specific, we did not consider the *technical means* by which new features will be added to a running system. This usually requires to bring the system to a clean state, where the code can be adapted, downloading the new code, before switching to the new configuration. This reconfiguration might require resources that temporarily decrease the performance of the system. Neither we did consider constraints on the implementation. For instance, a *distributed* implementation might be required [29]. Finally, our current algorithms only provides any strategy satisfying the requirements, but no notion of *preference* among strategies is introduced. In particular, one could prefer strategies that use less memory, require less computation, or minimize some user-defined cost.

Many systems operate in a continuous physical environment. *Hybrid models* can be used to integrate the modelling of the continuous and discrete parts [48, 21]. Furthermore, a realistic strategy for a hybrid system must be robust, i.e. able to cope with small disturbances [42]. A first step we did in this direction is to incorporate *real-time* [49, 43, 20].

In summary, we hope to complement our approach with others, as needed by the vast and multi-disciplinary area of self-adaptive systems, that are expected to progressively enter all domains [32], starting with controllers for space [12], networked systems [35, 24], robotics [53], anti-intrusion systems [45], cloud computing [52, 23], etc. where the need is most pressing.

References

1. R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Proceedings of FASE '98*, pages 21–37, Lisbon, Portugal, March 1998.
2. R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, Sept. 2002.
3. R. Alur, T. A. Henzinger, O. Kupferman, and M. Y. Vardi. Alternating refinement relations. In *Proceedings of CONCUR '98*, pages 163–178, London, UK, 1998. Springer-Verlag.
4. S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Feature-interaction detection using feature-aware verification. In *Proceedings of ASE'11*, pages 372–375. IEEE Computer Society, 2011.
5. A. Arnold, A. Vincent, and I. Walukiewicz. Games for synthesis of controllers with partial observation. *Theoretical computer science*, 303(1):7–34, 2003.

6. P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi. A logical framework to deal with variability. In *Proceedings of IFM'10*, pages 43–58, Berlin, Heidelberg, 2010. IEEE.
7. P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi. Formal description of variability in product families. In *Proceedings of SPLC'11*, pages 130–139. Springer-Verlag, 2011.
8. F. Bachmann, M. Goedicke, J. C. S. do Prado Leite, R. L. Nord, K. Pohl, B. Ramesh, and A. Vilbig. A meta-model for representing variability in product family development. In *Proceedings of PFE '03*, pages 66–80, 2003.
9. D. Bresolin and M. Capiluppi. A game-theoretic approach to fault diagnosis and identification of hybrid systems. *Theoretical Computer Science*, 2012. To appear.
10. M. Calder, M. Kolberg, E. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.
11. K. Chatterjee, L. Doyen, and T. Henzinger. A survey of partial-observation stochastic parity games. *Formal Methods in System Design*, pages 1–17, 2012.
12. W. Chen and M. Saif. Observer-based fault diagnosis of satellite systems subject to time-varying thruster faults. *Journal of dynamic systems, measurement, and control*, 129(3):352–356, 2007.
13. B. H. C. Cheng and al. Software engineering for Self-Adaptive systems: A research roadmap. In B. H. C. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, chapter 1, pages 1–26. Springer, Berlin Heidelberg, 2009.
14. R. Cieslak, C. Desclaux, A. Fawaz, and P. Varaiya. Supervisory control of discrete-event processes with partial observations. *Automatic Control, IEEE Transactions on*, 33(3):249–260, 1988.
15. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.
16. A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. In *Proceedings of ICSE'11*, pages 321–330. ACM, 2011.
17. A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *Proceedings of ICSE '10*, pages 335–344, New York, NY, USA, 2010. ACM.
18. M. Cordy, A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Managing evolution in software product lines : A model-checking perspective. In *Proceedings of VaMoS'12*, pages 183–191. ACM, 2012.
19. M. Cordy, A. Classen, G. Perrouin, P. Heymans, P.-Y. Schobbens, and A. Legay. Simulation-based abstractions for software product-line model checking. In *Proceedings of ICSE'12*. IEEE, 2012.
20. M. Cordy, P. Schobbens, P. Heymans, and A. Legay. Behavioural modelling and verification of real-time software product lines. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 66–75. ACM, 2012.
21. J. Cury, B. Krogh, and T. Niinomi. Synthesis of supervisory controllers for hybrid systems based on approximating automata. *Automatic Control, IEEE Transactions on*, 43(4):564–568, 1998.
22. K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *Proceedings of GPCE '05*, pages 422–437, 2005.

23. E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15(3):313–341, 2008.
24. S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli. A survey of autonomic communications. *ACM Trans. Auton. Adapt. Syst.*, 1(2):223–259, Dec. 2006.
25. C. E. Ebeling. *An introduction to reliability and maintainability engineering*. McGraw-Hill, 1997.
26. A. Fantechi and S. Gnesi. Formal modeling for product families engineering. In *SPLC*, pages 193–202, 2008.
27. A. Filieri, C. Ghezzi, and G. Tamburrelli. Run-time efficient probabilistic model checking. In *Proceedings of ICSE'11*, pages 341–350, 2011.
28. E. Filiot, N. Jin, and J.-F. Raskin. Antichains and compositional algorithms for ltl synthesis. *Formal Methods in System Design*, 39:261–296, 2011. 10.1007/s10703-011-0115-3.
29. B. Finkbeiner and S. Schewe. Uniform distributed synthesis. In *Logic in Computer Science, 2005. LICS 2005. Proceedings. 20th Annual IEEE Symposium on*, pages 321–330. IEEE, 2005.
30. D. Fischbein, S. Uchitel, and V. Braberman. A foundation for behavioural conformance in software product line architectures. In *Proceedings of ROSATEA'06*, pages 39–48. ACM Press, 2006.
31. N. Francez and I. R. Forman. Superimposition for interacting processes. In *Proceedings of CONCUR'90*, volume 458 of *LNCS*, pages 230–245. Springer, 1990.
32. H. Giese and B. H. C. Cheng, editors. *SEAMS '11: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, New York, NY, USA, 2011. ACM.
33. H. Gomaa and M. Hussein. Dynamic software reconfiguration in software product families. In F. van der Linden, editor, *Proceedings of PFE '03*, volume 3014 of *Lecture Notes in Computer Science*, pages 435–444. Springer Berlin / Heidelberg, 2004.
34. N. Griffeth, Y.-J. Lin, and al. *Feature interactions in telecommunications and software systems (FIW, ICFI)*. Ios Press, 1992-2012.
35. T. Gross and H. Sayama. *Adaptive Networks: Theory, Models and Applications*. Understanding Complex Systems. Springer, 2009.
36. A. Gruler, M. Leucker, and K. Scheidemann. Modeling and model checking software product lines. In *Proceedings of FMOODS'08*, pages 113–131. Springer, 2008.
37. G. Kalyon, T. Le Gall, H. Marchand, and T. Massart. Symbolic supervisory control of infinite transition systems under partial observation using abstract interpretation. *Discrete Event Dynamic Systems*, pages 1–41, 2012.
38. J. Kramer and J. Magee. Analysing dynamic change in software architectures: A case study. In *Proceedings of the International Conference on Configurable Distributed Systems*, Proceedings of CDS '98, pages 91–100, Washington, DC, USA, 1998. IEEE Computer Society.
39. S. Kulkarni and K. Biyani. Correctness of component-based adaptation. In I. Crnkovic, J. Stafford, H. Schmidt, and K. Wallnau, editors, *Proceedings of CBSE '04*, volume 3054 of *Lecture Notes in Computer Science*, pages 48–58. Springer Berlin / Heidelberg, 2004.
40. H. C. Li, S. Krishnamurthi, and K. Fisler. Interfaces for modular feature verification. In *Proceedings of ASE'02*, pages 195–204, 2002.
41. H. C. Li, S. Krishnamurthi, and K. Fisler. Verifying cross-cutting features as open systems. In *SIGSOFT FSE*, pages 89–98, 2002.

42. R. Majumdar, E. Render, and P. Tabuada. Robust discrete synthesis against unspecified disturbances. In *Proceedings of the 14th international conference on Hybrid systems: computation and control*, HSCC '11, pages 211–220, New York, NY, USA, 2011. ACM.
43. O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *STACS 95*, pages 229–242. Springer, 1995.
44. A. Paoli, M. Sartini, and S. Lafortune. Active fault tolerant control of discrete event systems using online diagnostics. *Automatica*, 47(4):639 – 649, 2011.
45. D. Ragsdale, C. Carver Jr, J. Humphries, and U. Pooch. Adaptation techniques for intrusion detection and intrusion response systems. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, volume 4, pages 2344–2349. IEEE, 2000.
46. M. SAMPATH, R. SENGUPTA, S. LAFORTUNE, K. SINNAMOHIDEEN, and D. TENEKETZIS. Diagnosability of discrete event system. *IEEE Transactions on Automatic Control*, 40(9):1555–1575, Sept. 1995.
47. P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Feature Diagrams: A Survey and A Formal Semantics. In *Proceedings of RE '06*, pages 139–148, 2006.
48. H. Wong-Toi. The synthesis of controllers for linear hybrid automata. In *Decision and Control, 1997., Proceedings of the 36th IEEE Conference on*, volume 5, pages 4607–4612. IEEE, 1997.
49. H. Wong-Toi and G. Hoffmann. The control of dense real-time discrete event systems. In *Decision and Control, 1991., Proceedings of the 30th IEEE Conference on*, pages 1527–1528. IEEE, 1991.
50. J. Zhang and B. H. Cheng. Using temporal logic to specify adaptive program semantics. *Journal of Systems and Software*, 79(10):1361 – 1369, 2006.
51. J. Zhang, H. J. Goldsby, and B. H. Cheng. Modular verification of dynamically adaptive systems. In *Proceedings of AOSD '09*, pages 161–172, New York, NY, USA, 2009. ACM.
52. Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010.
53. C. Zhong and S. A. DeLoach. Runtime models for automatic reorganization of multi-robot systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 20–29, New York, NY, USA, 2011. ACM.